REMARKS/ARGUMENTS

Applicants have received the Office action dated June 20, 2005, in which the Examiner: 1) rejected claims 1-30 and 33 under 35 U.S.C. § 112, second paragraph; and 2) rejected claims 1-36 under 35 U.S.C. § 102(a) as being unpatentable over Rotenberg ("AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors"). With this Response, Applicants have amended claims 1, 10, 19, and 31-33.

The Examiner's comments regarding the § 112, second paragraph, rejections of claims 1, 10 and 19 are as follows:

> [I]t is unclear how a main program can be running and spawn a pre-execution thread and have that pre-execution thread run ahead of the that currently running program (i.e. how is the main program which was running first ever be running behind a program which was created after the main program?).

Office action, page 2.

It may be that the Examiner has not fully understood Applicants' contribution. This issue is also relevant to the prior art rejections. The following discussion is intended to clarify Applicants' contribution and should not be used to limit the scope of the claims; the claims speak for themselves.

Figure 3 is illustrative of Applicants' contribution and is well-explained in Applicants' disclosure, repeated below for convenience.

> Figure 3 shows the same portion of a program in two different forms 202 and 220. Each program form 202, 220 includes two program segments 204 and 206. Program 202 represents the program without the ability to spawn pre-execution threads. Program 220 represents program 202 modified to spawn pre-execution threads at desired points during program execution. As shown, the modifications include start and stop pre-execution thread instructions 222 and 226, as well as a label 224. In accordance with the preferred embodiment, the instruction to start a pre-execution thread is of the form "PreExecute_Start(label)" where "label" is a value that informs the program counter 106 where to begin executing the pre-execution thread. In the example of Figure 3, "label" is "LIST2" which is identified by numeral 224. Thus, at run-time, when PreExecute_Start(LIST2) is executed by the main thread, a pre-execution thread will be spawned to execute the program starting at the label LIST2. The pre-execution thread then continues executing

> code segment 206 from LIST2 until the PreExecute_Stop() instruction 226 is encountered.
>
> While the main thread is executing code segment 204, the pre-execution thread runs ahead of the main thread and executes code segment 206. Cache misses encountered during the execution of segment 206 by the pre-execution thread preferably are resolved before the main thread encounters the same memory references when it executes code segment 206. Broadly speaking, the pre-execution thread resolves one or more memory references and causes the requested data to be in data cache 146 before the main thread needs the data.

Applicants' disclosure page 9.

Applicants thus clearly explain that code segment 206 (which is a portion of a program as shown) executes in a pre-execution thread and in the main thread, but begins execution earlier in the pre-execution thread than in the main thread. Thus, the main thread spawns a pre-execution thread to begin executing a portion of the code that the main thread will execute in the future, but has not yet started executing. In at least some embodiments, the pre-execution thread does not execute the entire program that is running in the main thread, but only select portions of the program that are specified by the main thread. Those portions that are selected for running in the pre-execution thread begin execution before their counterparts in the main thread begin execution.

After reviewing the comments above, the Examiner should find that claims 1, 10 and 19 are indeed clear and meet the requirements of § 112, second paragraph. If the Examiner is still confused, Applicants would be happy to discuss this further over the telephone.

The Examiner rejected claim 5 under § 112, second paragraph, for including the reference "said processor determines whether sufficient hardware resources are available before spawning said pre-execution thread." The Examiner stated that "it is uncertain as to how the processor 'determines whether sufficient hardware resources' are available (i.e., upon what criteria does the processor make this determination as to whether or not to spawn a process?)." Applicants believe claim 5 meets the requirements of § 112, second paragraph, which only requires that the specification concludes with one or more claims

"particularly pointing out and distinctly claiming the subject matter which the applicant regards as his invention." This statute does not require Applicants to explain in detail how to implement the claimed invention. Further, Applicants believe one of ordinary skill in the art would understand what claim 5 covers which is simply that the "processor determines whether sufficient hardware resources are available before spawning said pre-execution thread." The Examiner asks what criteria should be used to make this determination. For purposes of claim 5, such detailed information is not necessary to comply with § 112, second paragraph.

As for claim 33, the Examiner asked the question: "what does the pre-execution thread do in order to 'spin' a predetermined value in relationship to threads to pre-execute?" Office action page 2. Applicants submit that the concept of "spinning on a variable" is a well-known term of art. See e.g., attached printouts of web pages in which "spinning" on a variable is discussed. Applicants thus believe this limitation to be clear to one of ordinary skill in the art upon reading Applicants' disclosure.

Applicants amended claim 1 in two regards. First, reference to the "I/O controller" and the "I/O device" have been removed as not being necessary to patentability. Second, claim 1 has been amended to clarify that the processor includes an instruction that spawns "a pre-execution thread in which only a portion, but not all, of the same program executes." Rotenberg does not disclose this limitation.

Rotenberg refers to a processor in which to instruction streams—an active (A) stream and a redundant (R) stream—are executed. See Figure 2 and associated text. That is, the same program is executed twice in its entirety through the same processor, once in the A stream and again in the R stream. Rotenberg does not disclose that the A stream includes an instruction that spawns the R stream. The Abstract of Rotenberg explains that the "program is duplicated and the two redundant programs simultaneously run on the processor." The A stream in Rotenberg does not include an instruction that causes "only a portion, but not all, of the same program" to be executed in the R

stream. For these reasons, claims 1 and all claims dependent thereon are allowable over Rotenberg.

Dependent claim 2 refers to a "start instruction" and a "stop instruction." In reference to Figure 3, the Examiner stated that Rotenberg discloses a "branch-start" instruction and a "branch-return" instruction. In Figure 3 Applicants find only a reference to the conditional branch instruction "branch i5, r3==0" which means that control should jump to instruction i5 if r3=0, otherwise control continues with the instruction immediately following the branch instruction. Applicants find no reference to branch-start or branch-return instructions in Figure 3, and moreover have never heard of such instructions. The Examiner is respectfully requested to clarify what branch-start and branch-return instructions are. Applicants believe Rotenberg fails to teach or even suggest the limitations of claim 2.

Applicants amend claim 10 to refer to "only a portion, but not all," of the same program. For at least the same reasons articulated above with regard to claim 1, claim 10 and all claims dependent therefrom are allowable. The comments provided above with regard to claim 2 also apply to dependent claim 11.

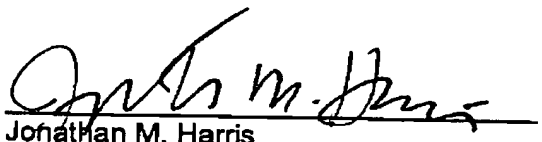
Claim 19 has been amended to refer to the "pre-execution thread running ahead of the main thread and containing only a portion of the instructions from the main thread." In Rotenberg, the A-stream and the R-stream are identical and thus the R-stream does not contain "only a portion of" the instructions from A-stream. For this reason, claim 19 and all claims dependent thereon are allowable over Rotenberg.

Applicants amended claim 31 to specify "wherein, in a pre-execution thread, said processor pre-executes instructions from a main thread that are specified by said main thread." Rotenberg does not teach or suggest the A-stream specifying which instructions are to be pre-executed. Thus, claim 31 and its dependent claims are allowable.

Applicants respectfully request reconsideration and that a timely Notice of Allowance be issued in this case. It is believed that no extensions of time or fees are required, beyond those that may otherwise be provided for in documents

accompanying this paper. However, in the event that additional extensions of time are necessary to allow consideration of this paper, such extensions are hereby petitioned under 37 C.F.R. § 1.136(a), and any fees required (including fees for net addition of claims) are hereby authorized to be charged to Hewlett-Packard Development Company's Deposit Account No. 08-2025.

Respectfully submitted,



Jonathan M. Harris
PTO Reg. No. 44,144
CONLEY ROSE, P.C.
(713) 238-8000 (Phone)
(713) 238-8008 (Fax)
ATTORNEY FOR APPLICANTS

HEWLETT-PACKARD COMPANY
Intellectual Property Administration
Legal Dept., M/S 35
P.O. Box 272400
Fort Collins, CO 80527-2400

Getting Ready for Hyper-Threading Technology Part 2: Thread Synchronization in Loops

Introduction

By Andrew Binstock

Hyper-Threading Technology is Intel's new technology for running two separate threads on the same processor. It is available today in Intel Xeon processors, and Intel has announced it will be available for desktop platforms in late 2002. This transition, I contend, will signal the start of a new era in software development in which desktop applications will be written as multithreaded applications, joining their server-side counterparts in this performance-oriented design.

Transitioning client-side applications from the established single-threaded model to a multithreaded design, however, requires several skills: a new way of looking at program construction and data, as well as familiarity with techniques of multithreaded programming.

In my last installment, I presented an overview of rethinking program design for multithreading. This change, frequently referred to as functional decomposition, requires not only new ways of assembling the building blocks, but also building blocks that are themselves different. I also examined data decomposition. This technique facilitates multithreaded design by placing emphasis on processing distinct blocks of data separately, so that they can run simultaneously on separate threads. I emphasized the importance of minimizing the dependence between blocks of data. And I articulated the essential rule of program and data design for multithreaded applications: minimize interactions and dependencies between threads.

In this installment, I discuss the types of problems and issues that occur when threads interact and how to handle some of them (not all situations can be handled perfectly). The type of interaction, the number of threads looking to share one resource, and the number of threads that can execute at once, are all factors that affect both the problem and its optimal solution. This article presents the predictable situations and recommendations for solution, as well as details specific to Hyper-Threading Technology. The goal is to design applications for Hyper-Threading Technology that immediately realize the performance advantages of the technology.

Thread Synchronization

Two threads in the same program rarely can operate on completely independent problems and distinct data sets. More often, threads have to interact—generally to synchronize activity. This can be as simple as polling a location in memory to see whether another thread has set a flag, or using common data structures like the instruction queue discussed in the last installment.

Spin-wait Loops

When a thread needs to wait for only a brief moment to synchronize with another thread, it typically uses a spin-wait loop. "Spin-wait" simply indicates a loop that is cycling—or spinning—while waiting for something else to happen. It generally takes the form of a null loop, as in the following C pseudo-code:

```
do { /* nothing */ }

  while ( synchronize_variable != desired_value

);
```

Spin-wait Loops

This code will spin endlessly until the two variables are equal; it then begins processing the next instruction. Effectively, this loop pauses the thread in which it is placed until another thread has synchronized by changing the value of **synchronize_variable**.

Spin-wait loops of this kind are fairly common, but on today's advanced processors, they cause unintended side-effects that must be examined. The first is that since the introduction of the Pentium® Pro processor, Intel processors perform speculative, out-of-order execution to accelerate instruction throughput. Instructions such as reading and comparing the variables are recognized by the processor as not being dependent on other instructions and so are candidates for out-of-order execution. When a processor performs speculative execution, it actually executes the instructions and then reintegrates the results into the original stream of executed processor instructions at the right place (a process known as retiring the instruction). In a loop as tight as this, the processor and the

speculative execution are both issuing instructions to read the same memory location. As a result, each instruction has to wait for the previous instruction to check the variable.

Needless to say, to have an execution pipeline stalled by a loop that is waiting on itself is hardly optimal. An additional problem occurs with very short tight loops: more than one instance of the loop can exist in the execution pipeline. In such a case, if the first instance exits the loop, instructions in the pipeline have to be flushed. Intel suggests solving both of these problems by adding the **pause** instruction to small tight loops. The new loop would look like:

```
do {

  _asm pause    ; this instruction
solves the problem

}

while ( synchronize_variable != desired_value );
```

It is counterintultive but true that the slight pause introduced by this new instruction accelerates processing. It allows the variable to be read at the maximum frequency the memory can sustain. Instructions are not stalled while waiting for previous ones to complete, and the processor is not tied up when it could make resources available for Hyper-Threading Technology. On architectures prior to the Intel NetBurst microarchitecture of the Pentium 4 and Intel Xeon processors, the **pause** instruction translates into a NOP (no operation) instruction.

Why doesn't this problem occur more often? In a normal instruction flow, the processor does not repeatedly perform the same instruction, hence instructions that are waiting for cache access are not all waiting to read the same location. Likewise, when the processor is tied up in execution, the speculative execution engine is executing other instructions and keeping the processor busy. In the case of the loop, the speculative execution is also reading the same memory location as the main processor, further compounding the backlog. In essence, processors today are designed to be kept busy executing instructions through a variety of means that all work well as long as they are not all accessing the same resource or memory location. When this occurs—and a tight loop is about the only time it will—the problem we've been discussing arises.

Synchronizing More Slowly

The technique of using a pause instruction is ideal when checking a synchronization variable that will change imminently. If the time it takes for the variable to change is greater than the time required for the operating system to switch contexts twice (once to a new context and once to switch back), it may make sense to use other approaches.

Spin loops, even when using the pause instruction, tie up computing resources and prevent other threads from running on the execution pipeline. Two options are available to the programmer to economize resource usage. One slows the loop, thereby consuming fewer processor resources and possibly releasing the execution pipeline to other threads; the other actively informs the operating system that the current thread is waiting and that the execution pipeline is temporarily available to another thread.

Slowing the Loop

An easy way to slow a loop is to use the **sleep()** function. This is an ANSI standard function that accepts the number of seconds to pause. Because seconds are far too coarse a unit of time for thread management, the Microsoft Win32* API uses the **Sleep()** function—note the initial capital letter—whose argument is the number of milliseconds to pause. The typical number of milliseconds is frequently in the single digits, although it can easily be more, depending on the event being waited for. Ideally, it should be the smallest value that does not waste cycles. Unfortunately, the best way to determine this value is empirically. Obtaining this value empirically highlights one of the difficulties of optimizing multithreaded applications. Code run on a two-processor machine might generate a different value than code run on a four-processor system. Use the value most likely to be found on the deployment platform for your software.

Sleep()need not be limited to synchronization. It can be used in straightforward polling of slow devices. Consider, for example, the keyboard. A very good typist can enter 120 words per minute at top speed, or two words per second. At an average of 5 characters per word, a key is being pressed every 100 milliseconds—an eternity for a processor. Tying up an execution pipeline polling the keyboard status is suboptimal if other threads need the processor. Insert a **Sleep()** statement into the body of the loop.

Putting a thread to sleep for a defined period of time immediately frees up resources on a Hyper-Threading Technology processor. Recall that in Hyper-Threading Technology, two logical processors are sharing the execution resources of a single chip. So, when one thread suspends operations, all the resources are immediately available to the other thread. These resources might instead be given

to a new thread.

Telling the Operating System

Ideally, when one thread has to wait a long time to synchronize with another, it is desirable to inform the operating system through a thread-blocking API. All modern operating systems have a library (generally derived from the POSIX thread library) that allows the program to block a thread that is entering a long wait and return those resources to the operating system for allocation to other threads.

Using the operating system to manage threads in this way is a good policy, and opportunities to involve the operating system are worth seeking out. Operating systems are generally well-tuned for thread management and can schedule threads efficiently. In situations where there are more threads executing than there are processors, using the operating system this way is almost imperative. The operating system is constantly juggling threads, so providing it with temporary resources to manage threads can make a significant difference in performance.

As with many aspects of tuning multithreaded applications, just when to use system calls is difficult to know exactly. Sometimes techniques have to be tried and tested to find the best possible mix. For example, **Sleep()** does not guarantee that the system will map another thread, nor does Windows* guarantee that the exact duration of the **Sleep()** argument will be obeyed. There is no way of knowing the best measures but to try and test. The reward, though, of snappy programs and happy users is certainly worth the extra effort. (For information on using Windows APIs to manage threads, see Multithreaded Programming In a Microsoft Win32 Environment, by Soumya Guptha.

Conclusion

This installment and the previous one have been a gentle introduction to the issues a developer faces when moving from a single-threaded mindset into the design and development of multithreaded programs. In the next installment, we will look at other issues, including low-level aspects of Hyper-Threading Technology.

Andrew Binstock is the principal analyst at Pacific Data Works LLC. He was previously a senior technology analyst at PricewaterhouseCoopers, and earlier editor in chief of *UNIX Review* and *C Gazette*. He is the lead author of "Practical Algorithms for Programmers," from Addison-Wesley Longman, which is currently in its 12th printing and in use at more than 30 computer-science departments in the United States. He can be reached at abinstock@pacificdataworks.com.

Windows Hardware & Driver Central
Microsoft Hardware News
February 17, 2004

» Windows Driver Development Tips

Using Spin Locks

A spin lock is a mechanism that kernel-mode driver developers can use to synchronize shared data between multiple threads in Microsoft® Windows®. While one thread owns a spin lock, any other thread that is waiting to acquire the lock "spins" on a memory location. When the spin lock is released, the waiting thread acquires the lock. A waiting thread is not suspended or paged out; it retains control of the CPU, thus preventing execution of other code at the same or a lower IRQL.

Spin locks are opaque objects of type KSPIN_LOCK. They must be allocated from nonpaged memory, such as the device extension of a driver-created device object or nonpaged pool allocated by the caller. In this discussion, we will look at three types of spin locks:

- Ordinary spin locks
- Queued spin locks
- Interrupt spin locks

All types of spin locks raise the IRQL to DISPATCH_LEVEL or higher. Spin locks are the only synchronization mechanism that can be used at IRQL >= DISPATCH_LEVEL. Code that holds a spin lock runs at IRQL >= DISPATCH_LEVEL, which means that the system's thread switching code (the dispatcher) cannot run and, therefore, the current thread cannot be pre-empted. Drivers should hold spin locks for only the minimum required amount of time and eliminate from the locked code path any tasks that do not require locking. Holding a spin lock for an unnecessarily long duration can negatively affects system performance.

IMPORTANT: All code within the spin lock must conform to the guidelines for running at IRQL >= DISPATCH_LEVEL. If code within the spin lock causes a page fault, the result is a system crash with the bug check value IRQL_NOT_LESS_OR_EQUAL. These system crashes occur because at IRQL >= DISPATCH_LEVEL, the operating system cannot wait for the kernel dispatcher event that is set internally when paging I/O completes. For complete information about these guidelines, see the white paper "Thread Context, Scheduling, and IRQL," which is available at http://www.microsoft.com/whdc/hwdev/driver/IRQL.mspx.

To implement spin locks on a single-processor system, the operating system has only to raise the IRQL, which prevents pre-emption of the current thread. Because no other threads can run concurrently, raising the IRQL is adequate to protect

Table of Contents

» Microsoft KB Articles

How to call video miniport driver functions from a display driver
http://support.microsoft.com/default.aspx?id=832517

Known issues with NDIS Intermediate (IM) driver samples
http://support.microsoft.com/default.aspx?id=323458

any shared structures. (Note, however, that the checked build of the operating system uses spin locks, even on single-processor systems.) On an SMP system, the operating system raises the IRQL and then spins by testing and setting a variable using an interlocked instruction.

Ordinary Spin Locks
Ordinary spin locks work at DISPATCH_LEVEL. To create an ordinary spin lock, a driver allocates a KSPIN_LOCK structure in nonpaged memory and then calls **KeInitializeSpinLock** to initialize it. Code that runs at IRQL < DISPATCH_LEVEL must acquire and release the lock by calling **KeAcquireSpinLock** and **KeReleaseSpinLock**. These routines raise the IRQL before acquiring the lock and then lower the IRQL upon release of the lock.

Code that is already running at IRQL = DISPATCH_LEVEL should call **KeAcquireSpinLockAtDpcLevel** and **KeReleaseSpinLockFromDpcLevel** instead. These routines do not change the IRQL.

Queued Spin Locks
Queued spin locks are a more efficient variation of ordinary spin locks. Queued spin locks are available in Microsoft Windows XP and later releases of Windows. Whenever multiple threads request the same queued spin lock, the waiting threads are queued in order of their request. In addition, queued spin locks test and set a variable that is local to the current CPU, so they generate less bus traffic and are more efficient on non-uniform memory architectures (NUMA).

A queued spin lock requires a KLOCK_QUEUE_HANDLE structure in addition to a KSPIN_LOCK structure. The KLOCK_QUEUE_HANDLE structure provides storage for a handle to the queue and the associated lock. This structure can be allocated on the stack. To initialize a queued spin lock, the driver calls **KeInitializeSpinLock**.

To ensure that the IRQL is properly raised and lowered, driver routines that run at PASSIVE_LEVEL or APC_LEVEL must call **KeAcquireInStackQueuedSpinLock** and **KeReleaseInStackQueuedSpinLock** to acquire and release these locks Driver routines that run at DISPATCH_LEVEL should call **KeAcquireInStackQueuedSpinLockAtDpcLevel** and **KeReleaseInStackQueuedSpinLockFromDpcLevel** instead. These routines do not raise and lower the IRQL.

Queued Spin Locks
An interrupt spin lock protects data such as device registers that a driver's *InterruptService* routine and *SynchCritSection* routine access at the specified device interrupt request level (DIRQL). When a device driver connects its interrupt object, the operating system creates an interrupt spin lock associated with that interrupt object. The driver is not required to allocate storage for the spin lock or to initialize it.

When an interrupt occurs, the system raises the IRQL on the processor to DIRQL for the interrupting device, acquires the default interrupt spin lock associated with the interrupt object, and then calls the driver's *InterruptService* routine. While the *InterruptService* routine is running, the processor IRQL remains at DIRQL and the operating system holds the corresponding interrupt spin lock. When the *InterruptService* routine exits, the operating system releases the lock and lowers the IRQL (unless another interrupt is pending at that level).

The system also acquires the default interrupt spin lock when a driver calls **KeSynchronizeExecution** to run a *SynchCritSection* routine. The operating system raises the IRQL to DIRQL for the device, acquires the lock, and invokes the *SynchCritSection* routine. When the routine exits, the operating system releases the lock and lowers the IRQL. Other driver routines that share data with the *InterruptService* routine or *SynchCritSection* routine must call **KeAcquireInterruptSpinLock** to acquire this lock before they can access the shared data. **KeAcquireInterruptSpinLock** is available on Windows XP and later releases of Windows.

Some types of devices can generate multiple interrupts at different levels. Examples include devices that support PCI 3.0 MSI-X, which generate message-signaled interrupts (MSI), and a few older devices that interrupt at more than one IRQL. Drivers that support such devices must serialize access to data among two or more *InterruptService* routines.

In this case, the driver must create a spin lock to protect shared data at the highest DIRQL at which any interrupt may arrive. When the driver connects its interrupt objects, it passes a pointer to the driver-allocated KSPIN_LOCK structure, along with the highest DIRQL at which the device interrupts. The operating system associates the driver-created spin lock and the DIRQL with the interrupt object.

When the operating system calls the *InterruptService* routines, it raises the IRQL to the DIRQL specified with the interrupt object and acquires the driver-created spin lock. The system also uses this lock when it runs a *SynchCritSection* routine. Other driver routines that share data with the *InterruptService* or *SynchCritSection* routine must call **KeAcquireInterruptSpinLock** to acquire this lock before they can access the shared data.

The next version of Microsoft Windows (codenamed "Longhorn") includes significant changes to the interrupt architecture to support message-signaled interrupts. Specifically, the **IoConnectInterrupt** routine is deprecated. You should use its replacement, **IoConnectInterruptEx** in new drivers, and older versions of drivers should be updated to use this new routine if possible.

For more information on these changes, see the white paper

"Interrupt Architecture Enhancements In Microsoft Windows Codenamed Longhorn" at:
http://www.microsoft.com/whdc/hwdev/bus/pci/msi.mspx

The preview (pre-beta) version of the Windows Longhorn Driver Kit (LDK) provides the **IoConnectInterruptEx** routine for use in Microsoft Windows 2000 and later releases of Windows.

To find out more about obtaining a copy of the LDK, see:
http://www.microsoft.com/whdc/hwdev/longhorn/default.mspx

Call to Action
Nearly every driver requires spin locks and these mechanisms by their very nature can cause performance bottlenecks. Follow these guidelines to improve performance:

- Test your Windows kernel-mode driver using the Driver Verifier tool (verifier.exe) in Microsoft Windows XP and later operating systems. Use Driver Verifier global counters to monitor IRQL raises and spin lock acquisitions.
- Use an in-stack queued spin lock instead of an ordinary spin lock whenever several components might frequently contend for the lock
- Hold each spin lock for the minimum amount of time necessary, particularly if the lock is frequently required by other code. For example, traversing a long, linked list in a linear order while holding a heavily used spin lock can cause a performance bottleneck.
- For more information about IRQL issues for drivers, see the white paper "Scheduling, Thread Context, and IRQL," at http://www.microsoft.com/whdc/hwdev/driver/IRQL.mspx
- Additional information and best practices for driver developers concerning spin locks, fast mutexes, **InterlockedXxx** and **ExInterlockedXxx** routines, and other synchronization mechanisms in Windows kernel-mode drivers can be found in the "Locks, Deadlocks, and Synchronization" white paper at http://www.microsoft.com/whdc/hwdev/driver/locks.mspx

» News

Microsoft iSCSI Software Initiator Version 1.03
An updated version of the Microsoft iSCSI Software Initiator package (version 1.03) is now available. The Microsoft iSCSI Software Initiator allows a Microsoft Windows-based computer to serve as an iSCSI initiator to connect to iSCSI targets on an Internet Protocol Storage Area Network (IP SAN). All iSCSI devices appear in Windows as a local disk and can be managed in Disk Administrator, as any other local disk.

Supported Operating Systems:

- Windows Server™ 2003
- Windows XP Service Pack 1
- Windows XP 64-Bit Edition
- Windows 2000 Service Pack 3 or later

The following fixes were made in Microsoft iSCSI Software Initiator Version 1.03 release.

- On SCSI command timeout or on BUS Reset initiator will send Task Management command and will reset the TCP connection. It will then re-login to the session.
- A PDU with invalid command sequence number was being sent upon bus reset.
- An invalid response for **SCSIOP_REPORT_LUNS** was causing the initiator to crash.
- A Task Management command was being sent with an invalid logical unit number (LUN).
- An invalid header was being sent in response to R2T request.
- Tag leak caused under low memory condition.
- Initiator does not assume that padding bytes in a PDU will be zero. It will use whatever data is in the padding bytes to compute CRC.
- Handle **IOCTL_SCSI_GET_CAPABILITIES** and **IOCTL_SCSI_GET_INQUIRY_DATA** by PDO in Windows 2000.

Call to Action
Download the appropriate Microsoft iSCSI v1.03 package for your operating system at:
http://www.microsoft.com/downloads/details.aspx?FamilyID=12cb3c1a-15d6-4585-b385-befd1319f825&DisplayLang=en

For more information on usage of the Microsoft iSCSI Software Initiator v1.03, please see the user's guide included with the package.

Come Together at WinHEC 2004
WinHEC is the single opportunity each year for the industry to work together on critical technology issues and future directions for Windows, and to interact with Microsoft engineers and other industry leaders.

At WinHEC 2004 in Seattle, Microsoft Chairman and Chief Software Architect Bill Gates will present the opening keynote about Microsoft investments and industry opportunities for the Microsoft Windows platform.

Register before March 22 and save $300!

For more information on the conference and instructions for registering, see:
http://www.microsoft.com/whdc/winhec

» About This Newsletter

The Microsoft Hardware Newsletter, a publication of Microsoft Windows Hardware and Driver Central, is a resource for developers who are creating new hardware and drivers for Microsoft Windows operating systems.

Did you get this newsletter from a friend? To subscribe to the Microsoft Hardware Newsletter, visit:
http://register.microsoft.com/regsys/decisionpoint.asp.